ICSC Qualification Round 2025

Philipp Kolassa

August 2025

Problem A: Neural Network Components

$w_{21}^{(1)}$: Weight	$\sum$: sum function	f: activation function
Red circle: input values	Yellow circle: bias	Green circle: output value
Box A: hidden layer	Box B: output layer	$\hat{y}$: prediction

Problem B: Cake Calculator

Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    int cakes;
    int flour_left;
    int sugar_left;
} CakeResult;

CakeResult cake_calculator(int flour, int sugar) {
    CakeResult retVal;
    retVal.cakes=(flour/100<sugar/50)? flour/100:sugar/50;
    retVal.flour_left=flour-(retVal.cakes*100);
    retVal.sugar_left=sugar-(retVal.cakes*50);
    return retVal;
}
```

Listing 1: Cake Calculator

Explanation

This code is intended to work with the code skeleton. The code skeleton will not be explained here.

To first determine the amount of cakes that can be produced, the program checks whether $\frac{\text{flour}}{100}$ or $\frac{\text{sugar}}{50}$ is smaller as you will need 100 portions of flour and 50 portions of sugar for one cake. The result for the amount of cakes will be set to the smaller value as there can't be more cake if there are no more ingredients (line 13). Afterwards the left over flour and sugar are calculated by subtracting 100 times the amount of cakes from the initial flour and 50 times the amount of cakes from the inital sugar. The result is returned in the form of a struct which has been defined in the code skeleton.

Problem C: School Messaging App

Question 1

If you have a character with a high probability in a data set it often makes sense to store or transmit this character using less bits to encode this character while you allocate more bits to characters with a low probability. For example consider the following characters with their probabilities in a data set:

Character	Probability
A	0.8
B	0.1
C	0.1

As there are 3 possible occuring characters an encoding with fixed length would require at least 2 bits per character ($2^2 = 4$). For a string $s =$ AAAAAAAABC of length 10 it would therefore need $10 \cdot 2 = 20$ bits to encode.

A code with variable code word length on the other side could assign a short code to the often occuring character A and longer codes to B and C. A possible code could look like this:

Character	Code in bits
A	1
B	00
C	01

Using this code we could achieve a size of 12 bits, considerably smaller than the fixed length variant. This would allow more and bigger messages to be sent within a certain data limit.

Question 2

The entropy describes the optimal average number of bits needed to encode a character of a specific character set. The entropy H can be calculated using the following formula where n is the amount of characters and p_i is the probability of occurence for each character:

$$
\begin{aligned}
H &= -\sum_{i=1}^{n} p_i \cdot \log_2(p_i) \\
&= -\Big(0.20 \cdot \log_2(0.20) + 0.15 \cdot \log_2(0.15) + 0.12 \cdot \log_2(0.12) + 0.10 \cdot \log_2(0.10) \\
&\quad + \ 0.08 \cdot \log_2(0.08) + 0.06 \cdot \log_2(0.06) + 0.05 \cdot \log_2(0.05) + 0.05 \cdot \log_2(0.05) \\
&\quad + \ 0.04 \cdot \log_2(0.04) + 0.03 \cdot \log_2(0.03) + 0.02 \cdot \log_2(0.02) + 0.10 \cdot \log_2(0.10)\Big) \\
&\approx 3.32
\end{aligned}
$$

Therefore the optimal average amount of bits needed to encode a character is approximately 3.32.

Question 3

To calculate the average code length of a character for our Fano code we multiply the probability of each character with its code length:

$$
\begin{aligned}
\bar{L} &= \sum_{i=1}^{n} p_i \cdot L_i \\
&= \ 0.20 \cdot 3 + 0.15 \cdot 3 + 0.12 \cdot 3 + 0.10 \cdot 4 + 0.08 \cdot 4 + 0.06 \cdot 4 \\
&\quad + 0.05 \cdot 3 + 0.05 \cdot 4 + 0.04 \cdot 4 + 0.03 \cdot 4 + 0.02 \cdot 4 + 0.10 \cdot 4 \\
&= 3.48
\end{aligned}
$$

Therefore the average code length of a character with our Fano code is 3.48 bits which is slightly more than the entropy of around 3.32 bits. It could still be improved, especially as (to name an example) the letter 'G' with a probability of 0.05 is encoded with 3 bits while a letter with a much higher probability of 0.10 like 'C' is encoded with 4 bits. This could be improved but our Fano code's average length is already only about 5 % higher than the entropy, which is quite good.

Problem D: Word Search Puzzle

Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <ctype.h>

#define MAX_WORDS 20
#define MAX_WORD_LENGTH 20
#define GRID_SIZE 10

// Clears the board and writes spaces in every position.
void clear_board(char (*board)[GRID_SIZE]) {
    for (int i=0; i<GRID_SIZE; i++) {
        for (int k=0; k<GRID_SIZE; k++) {
            board[i][k]=' ';
        }
    }
}

// Fills empty positions with random letters.
void fill_board(char (*board)[GRID_SIZE]) {
    for (int y=0; y<GRID_SIZE; y++) {
        for (int x=0; x<GRID_SIZE; x++) {
            if (board[y][x]==' ') {
                board[y][x]=(char) (rand()%26)+97;
            }
        }
    }
}

// Writes a word into the board array in a specified direction.
int place_word(char (*board)[GRID_SIZE], char *word, int *position,
    int *direction) {
    int x_coordinate=position[0];
    int y_coordinate=position[1];

    for (int letter=0; word[letter]!='\0'; letter++) {
        board[y_coordinate][x_coordinate]=word[letter];
        x_coordinate+=direction[0];
        y_coordinate+=direction[1];
    }
    return 0;
}

// Checks whether there is space for a word starting at a position and
    going towards a certain direction.
int check_direction(char (*board)[GRID_SIZE], char *word, int *position,
    int *direction) {
    int x_coordinate=position[0];
```

```
49      int y_coordinate=position[1];
50
51      for (int letter=0; word[letter]!='\0'; letter++) {
52          if (x_coordinate<0 || x_coordinate>=GRID_SIZE || y_coordinate<0
53             || y_coordinate>=GRID_SIZE) {
54              return 0;
55          } else {
56              if (board[y_coordinate][x_coordinate]!=word[letter]
57                  && board[y_coordinate][x_coordinate]!=' ') {
58                  // the word doesn't fit if a position is already filled
59                  // with a different letter
60                  return 0;
61              }
62          }
63          x_coordinate+=direction[0];
64          y_coordinate+=direction[1];
65      }
66      return 1;
67  }
68
69  // Gives an array with all starting positions for the word. One will be
        chosen at random to prevent repeating patterns
70  void *init_positioning_array() {
71      int (*arr)[2]=malloc(GRID_SIZE*GRID_SIZE*2*sizeof(int));
72      for (int x=0; x<GRID_SIZE; x++) {
73          for (int y=0; y<GRID_SIZE; y++) {
74              arr[y+GRID_SIZE*x][0]=x;
75              arr[y+GRID_SIZE*x][1]=y;
76          }
77      }
78      return arr;
79  }
80
81  // Returns an array of size 2 where [0] corresponds to the x translation
        and [1] to the y translation for the directions of a word.
82  void *init_direction_array() {
83      // 8 directions, 2 coordinates
84      int (*arr)[2]=malloc(8*2*sizeof(int));
85      arr[0][0]=1, arr[0][1]=0;
86      arr[1][0]=1, arr[1][1]=-1;
87      arr[2][0]=0, arr[2][1]=-1;
88      arr[3][0]=-1, arr[3][1]=-1;
89      arr[4][0]=-1, arr[4][1]=0;
90      arr[5][0]=-1, arr[5][1]=1;
91      arr[6][0]=0, arr[6][1]=1;
92      arr[7][0]=1, arr[7][1]=1;
93      return arr;
94  }
95
96  // Processes a word from finding a position and direction to placing it
        into the grid array.
97  void process_word(char (*board)[GRID_SIZE], char *word) {
98      int (*pos_array)[2]=init_positioning_array();
```

```
99      int positions_checked=0;
100     int word_placed=0;
101
102     while (positions_checked<GRID_SIZE*GRID_SIZE) {
103         // random position of the ones, which are left
104         int rand_pos=rand()%(GRID_SIZE*GRID_SIZE-positions_checked);
105         // skipping over already checked spots
106         for (int pos_ptr; pos_ptr<=rand_pos; pos_ptr++) {
107             if (pos_array[pos_ptr][0]==-1) {
108                 rand_pos++;
109             }
110         }
111
112         int* checking_at_pos=pos_array[rand_pos];
113         int (*directions)[2]=init_direction_array();
114
115         int directions_checked=0;
116         while (directions_checked<8 && !word_placed) {
117             int rand_direction=rand()%(8-directions_checked);
118             for (int direction_ptr; direction_ptr<=rand_direction;
119                  direction_ptr++) {
120                 if (directions[direction_ptr][0]==-2) {
121                     // skipping already checked directions by increasing
122                     // the border by 1
123                     rand_direction++;
124                 }
125             }
126             if (check_direction(board, word, checking_at_pos,
127                 directions[rand_direction])) {
128                 place_word(board, word, checking_at_pos,
129                     directions[rand_direction]);
130                 word_placed=1;
131             } else {
132                 directions_checked++;
133                 // set the x value of the checked direction to -2 to skip
134                 // it in the next run
135                 directions[rand_direction][0]=-2;
136             }
137         }
138
139         free(directions);
140
141         if (word_placed) {
142             break;
143         } else {
144             positions_checked++;
145             pos_array[rand_pos][0]=-1;
146         }
147     }
148
149     free(pos_array);
150     if (!word_placed) {
151         printf("Error during position checking: No space for %s",word);
```

```
152     }
153 }
154
155 /**
156  * Generate a 10x10 word search puzzle containing the given words.
157  *
158  * @param words A vector of strings to include in the puzzle.
159  * @return A 2D vector of chars representing the word search puzzle.
160  */
161 void create_crossword(char words[][MAX_WORD_LENGTH], int wordCount, char
        grid[GRID_SIZE][GRID_SIZE]) {
162     srand(time(NULL));
163     clear_board(grid);
164
165     for (int w=0; w<wordCount; w++) {
166         process_word(grid, words[w]);
167     }
168     fill_board(grid);
169 }
```

Listing 2: Word Search Puzzle

Explanation

This code functions as the calculation part of the problem while the code skeleton's `main` function does the input and output handling. The code skeleton will not be explained here.

The `create_crossword` function gets an array of char arrays with all the words to be included in the puzzle as well as the amount of words and the two dimensional playing field array as parameters. It will first set the random seed using the time to ensure a (mostly) random position and direction of words. It then clears the board of preexisting data using the `clear_board` function which writes a space in every position of the board. The program will then iterate over the words and will execute the `process_word` function for each one, which in the end integrates a word given as a parameter into the board. After all words have been placed, the `create_crossword` function fills the remaining empty positions which have not been filled yet by any words using the `fill_board` function by inserting random ASCII lower case characters. This concludes the algorithm.

We will now take a look at the `process_word` function. This function takes the board and a word as parameters and is tasked with finding a spot and direction for this word on the board and in the end placing it. It starts by defining a positioning array (`pos_array`) using the `init_positioning_array` function. This array contains two-element arrays with the x and y values of all positions on the board (so 100 in total).

The benefit of using an array to access the positions instead of randomly generating them directly is that if there are no positions the word can fit into the board, it will not randomly keep looking but instead print an error and continue with the next word. This is also helpful as if there is only for example one possible location the word can go

it is bound to check this position after at worst 99 other checks while with randomly generating locations it might get stuck and never get to it.

We also define the variable `word_placed`, which is initially set to 0, to indicate when we have placed the word into a correct position. To keep track of how many positions have been checked, we lastly also define `positions_checked` and do a loop while this variable is below the total number of positions on the board.

In this loop, we now need to check whether a word can fit into a position. To find a random position to be checked in this run of the loop we generate a random number between 0 and the number of all positions minus the number of already checked ones. This is not the direct index in the `pos_array` as we do not have a good way to remove the checked positions from an array so the checked positions have an x value of -1. To therefore determine the index of the position to be checked in the array we iterate over the positions until we are at our random value and for every index below that we check whether this positions x value is -1. If it is we increase our random value by 1. In the end we will have the index of our position in our random value. We have to do this procedure to get the index as blatantly guessing could lead to the same problem of being stuck and just taking the first position without a x value of -1 would lead to reoccurring positioning patterns over the course of multiple games.

To sum everything up so far: We have a word that needs to be placed into the board. We also now have a position to check at. Therefore we now do basically the same for the direction as we did for the position. We define an array `directions` which we initialise using the `init_direction_array` function. This also is a two dimensional array of all possible directions (I chose to go with both diagonal and straight so a total of 8 directions) which has an x and y value in the sub-arrays which describe how the values of the position need to be modified to go to the position of the next letter. We also do the same procedure with the directions checked to prevent getting stuck and also generate a random value, which we modify to get an index of `directions` to ensure randomness.

To actually then check the direction we use the `check_direction` function. It takes the board, the word, the starting position of the word and the direction as parameters and returns 1 if the combination of these parameters works and the word can or returns 0 if it cannot be placed. To check for this the program iterates over the indices of the word and checks at each index whether there already is a character at the current position which is not the right character needed. Before it moves on to the next index it modifies the position to be checked using the x and y values from the direction array. If there ever is a position filled with a different character or if the position to be checked is outside the grid it returns 0. If it runs without any interruption it returns 1.

We now move back to the `process_word` function. If the `check_direction` function returns 0 the word can't be placed there so the `directions_checked` variable increases by 1 and the x value of this direction in the array `directions` is set to -2 to disable this index (it can't be set to -1 like with the positions as -1 is a possible value). If `check_direction` on the other hand returns 1 the `word_placed` variable is set to 1 to interrupt the loop in the next check and the `place_word` function is called which works

in the same way as the `check_direction` does except that it places the letters on the board and does not do any more checks.

When the while loop for the directions is done the program checks whether the word has been placed (is `word_placed==1`?). If it has, the while loop for the positions will be interrupted. If it has not, the word can't be placed in this position so this position is disabled by setting its x value in the `pos_array` to -1 and `positions_checked` is increased by 1.

When the while loop for the positions is done the program checks again whether the word has been placed. If it still has not, the program outputs an error message saying that the word could not be placed due to lack of space. This concludes the `process_word` function.

The solution described is elegant in a way because it places a word in constant run time and space complexity ($\mathcal{O}(1)$). It also does this while keeping a random approach, preventing reoccurring patterns and favouritism towards a position or direction. The entire algorithm therefore has a runtime of $\mathcal{O}(n)$ with respect to the amount of words. All of this is assuming a constant grid size. With a variable grid size the runtime would be $\mathcal{O}(n \cdot \texttt{GRID\_SIZE}^2)$. The algorithm could be further improved by optimising the positioning of words in the case of lack of space for a word on the board but I decided not to go this step as it would not be of any use in most cases and because I think that the puzzle will get too cramped with words.

Problem E: Functional Completeness of NAND

We can prove the functional complenetess of NAND by proving that our boolean base (AND, OR, NOT), which is functionally complete, can be constructed using NAND. Therefore we need to prove that we can construct each element of our boolean base with NAND. We will prove this using truth tables.

A	$\neg A$	A NAND A
0	1	1
1	0	0

Table 1: Representation of NOT

A	B	$A \wedge B$	$(A$ NAND $B)$ NAND $(A$ NAND $B)$
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

Table 2: Representation of AND

A	B	$A \vee B$	$(A$ NAND $A)$ NAND $(B$ NAND $B)$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

Table 3: Representation of OR

As the stated boolean expressions using NAND have the same values for all combinations of A and B as the ones using our boolean base, we have proven that we can represent all boolean expressions using AND, OR and NOT with boolean expressions using only NAND. As our boolean base is functionally complete we can therefore also say that NAND is functionally complete.